程序 12-13 linux/fs/open.c

```
1  /*
2   *  linux/fs/open.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
9  #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符操作控制常数符号定义。
10 #include <sys/types.h>    // 类型头文件。定义基本的系统和文件系统统计信息结构和类型。
11 #include <utime.h>        // 用户时间头文件。定义访问和修改时间结构以及 utime()原型。
12 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和符号常量等。
13
14 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
15 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
16 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
17
18 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
19
   //// 取文件系统信息。
   // 参数 dev 是含有已安装文件系统的设备号。ubuf 是一个 ustat 结构缓冲区指针，用于存放
   // 系统返回的文件系统信息。该系统调用用于返回已安装（mounted）文件系统的统计信息。
   // 成功时返回 0，并且 ubuf 指向的 ustate 结构被添入文件系统总空闲块数和空闲 i 节点数。
   // ustat 结构定义在 include/sys/types.h 中。
20 int sys_ustat(int dev, struct ustat * ubuf)
21 {
22         return -ENOSYS;                 // 出错码：功能还未实现。
23 }
24
   //// 设置文件访问和修改时间。
   // 参数 filename 是文件名，times 是访问和修改时间结构指针。
   // 如果 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。
   // 如果 times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。
25 int sys_utime(char * filename, struct utimbuf * times)
26 {
27         struct m_inode * inode;
28         long actime,modtime;
29
   // 文件的时间信息保存在其 i 节点钟。因此我们首先根据文件名取得对应 i 节点。如果没有找
   // 到，则返回出错码。如果提供的访问和修改时间结构指针 times 不为 NULL，则从结构中读取
   // 用户设置的时间值。否则就用系统当前时间来设置文件的访问和修改时间。
30         if (!(inode=namei(filename)))
31                 return -ENOENT;
32         if (times) {
33                 actime = get_fs_long((unsigned long *) &times->actime);
34                 modtime = get_fs_long((unsigned long *) &times->modtime);
35         } else
36                 actime = modtime = CURRENT_TIME;
   // 然后修改 i 节点中的访问时间字段和修改时间字段。再设置 i 节点已修改标志，放回该 i 节
   // 点，并返回 0。
37         inode->i_atime = actime;
38         inode->i_mtime = modtime;
```

```
39          inode->i_dirt = 1;
40          iput(inode);
41          return 0;
42  }
43
44  /*
45   * XXX should we use the real or effective uid?  BSD uses the real uid,
46   * so as to make this call useful to setuid programs.
47   */
    /*
     * XXX 我们该用真实用户 id（ruid）还是有效用户 id（euid）？BSD 系统使用了
     * 真实用户 id，以使该调用可以供 setuid 程序使用。
     * （注：POSIX 标准建议使用真实用户 ID）。
     * （注 1：英文注释开始的 'XXX' 表示重要提示）。
     */
    //// 检查文件的访问权限。
    // 参数 filename 是文件名，mode 是检查的访问属性，它有 3 个有效比特位组成：R_OK(值 4)、
    // W_OK(2)、X_OK(1) 和 F_OK(0) 组成，分别表示检测文件是否可读、可写、可执行和文件是
    // 否存在。如果访问允许的话，则返回 0，否则返回出错码。
48  int sys_access(const char * filename,int mode)
49  {
50          struct m_inode * inode;
51          int res, i_mode;
52
    // 文件的访问权限信息也同样保存在文件的 i 节点结构中，因此我们要先取得对应文件名的 i
    // 节点。检测的访问属性 mode 由低 3 位组成，因此需要与上八进制 0007 来清除所有高比特位。
    // 如果文件名对应的 i 节点不存在，则返回没有许可权限出错码。若 i 节点存在，则取 i 节点
    // 钟文件属性码，并放回该 i 节点。另外，57 行上语句“iput(inode);”最后放在 61 行之后。
53          mode &= 0007;
54          if (!(inode=namei(filename)))
55                  return -EACCES;                     // 出错码：无访问权限。
56          i_mode = res = inode->i_mode & 0777;
57          iput(inode);
    // 如果当前进程用户是该文件的宿主，则取文件宿主属性。否则如果当前进程用户与该文件宿
    // 主同属一组，则取文件组属性。否则，此时 res 最低 3 比特是其他人访问该文件的许可属性。
    // [[?? 这里应 res >>3 ??]
58          if (current->uid == inode->i_uid)
59                  res >>= 6;
60          else if (current->gid == inode->i_gid)
61                  res >>= 6;
    // 此时 res 的最低 3 比特是根据当前进程用户与文件的关系选择出来的访问属性位。现在我们
    // 来判断这 3 比特。如果文件属性具有参数所查询的属性位 mode，则访问许可，返回 0
62          if ((res & 0007 & mode) == mode)
63                  return 0;
64          /*
65           * XXX we are doing this test last because we really should be
66           * swapping the effective with the real user id (temporarily),
67           * and then calling suser() routine.  If we do call the
68           * suser() routine, it needs to be called last.
69           */
            /*
             * XXX 我们最后才做下面的测试，因为我们实际上需要交换有效用户 ID 和
             * 真实用户 ID（临时地），然后才调用 suser()函数。如果我们确实要调用
```

```
         * suser()函数，则需要最后才被调用。
         */
    // 如果当前用户 ID 为 0（超级用户）并且屏蔽码执行位是 0 或者文件可以被任何人执行、搜
    // 索，则返回 0。否则返回出错码。
70      if ((!current->uid) &&
71          (!(mode & 1) || (i_mode & 0111)))
72              return 0;
73      return -EACCES;                         // 出错码：无访问权限。
74  }
75
    //// 改变当前工作目录系统调用。
    // 参数 filename 是目录名。
    // 操作成功则返回 0，否则返回出错码。
76  int sys_chdir(const char * filename)
77  {
78      struct m_inode * inode;
79
    // 改变当前工作目录就是要求把进程任务结构的当前工作目录字段指向给定目录名的 i 节点。
    // 因此我们首先取目录名的 i 节点。如果目录名对应的 i 节点不存在，则返回出错码。如果该
    // i 节点不是一个目录 i 节点，则放回该 i 节点，并返回出错码。
80      if (!(inode = namei(filename)))
81              return -ENOENT;                 // 出错码：文件或目录不存在。
82      if (!S_ISDIR(inode->i_mode)) {
83              iput(inode);
84              return -ENOTDIR;                // 出错码：不是目录名。
85      }
    // 然后释放进程原工作目录 i 节点，并使其指向新设置的工作目录 i 节点。返回 0。
86      iput(current->pwd);
87      current->pwd = inode;
88      return (0);
89  }
90
    //// 改变根目录系统调用。
    // 把指定的目录名设置成为当前进程的根目录'/'。
    // 如果操作成功则返回 0，否则返回出错码。
91  int sys_chroot(const char * filename)
92  {
93      struct m_inode * inode;
94
    // 该调用用于改变当前进程任务结构中的根目录字段 root，让其指向参数给定目录名的 i 节点。
    // 如果目录名对应的 i 节点不存在，则返回出错码。如果该 i 节点不是目录 i 节点，则放回该
    // i 节点，也返回出错码。
95      if (!(inode=namei(filename)))
96              return -ENOENT;
97      if (!S_ISDIR(inode->i_mode)) {
98              iput(inode);
99              return -ENOTDIR;
100     }
    // 然后释放当前进程的根目录 i 节点，并重新设置为指定目录名的 i 节点，返回 0。
101     iput(current->root);
102     current->root = inode;
103     return (0);
104 }
```

```
105
    //// 修改文件属性系统调用。
    // 参数 filename 是文件名，mode 是新的文件属性。
    // 若操作成功则返回 0，否则返回出错码。
106 int sys_chmod(const char * filename,int mode)
107 {
108         struct m_inode * inode;
109
    // 该调用为指定文件设置新的访问属性 mode。文件的访问属性在文件名对应的 i 节点中，因此
    // 我们首先取文件名对应的 i 节点。如果 i 节点不存在，则返回出错码（文件或目录不存在）。
    // 如果当前进程的有效用户 id 与文件 i 节点的用户 id 不同，并且也不是超级用户，则放回该
    // 文件 i 节点，返回出错码（没有访问权限）。
110         if (!(inode=namei(filename)))
111                 return -ENOENT;
112         if ((current->euid != inode->i_uid) && !suser()) {
113                 iput(inode);
114                 return -EACCES;
115         }
    // 否则就重新设置该 i 节点的文件属性，并置该 i 节点已修改标志。放回该 i 节点，返回 0。
116         inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
117         inode->i_dirt = 1;
118         iput(inode);
119         return 0;
120 }
121
    //// 修改文件宿主系统调用。
    // 参数 filename 是文件名，uid 是用户标识符(用户 ID)，gid 是组 ID。
    // 若操作成功则返回 0，否则返回出错码。
122 int sys_chown(const char * filename,int uid,int gid)
123 {
124         struct m_inode * inode;
125
    // 该调用用于设置文件 i 节点中的用户和组 ID，因此首先要取得给定文件名的 i 节点。如果文
    // 件名的 i 节点不存在，则返回出错码（文件或目录不存在）。如果当前进程不是超级用户，
    // 则放回该 i 节点，并返回出错码（没有访问权限）。
126         if (!(inode=namei(filename)))
127                 return -ENOENT;
128         if (!suser()) {
129                 iput(inode);
130                 return -EACCES;
131         }
    // 否则我们就用参数提供的值来设置文件 i 节点的用户 ID 和组 ID，并置 i 节点已经修改标志，
    // 放回该 i 节点，返回 0。
132         inode->i_uid=uid;
133         inode->i_gid=gid;
134         inode->i_dirt=1;
135         iput(inode);
136         return 0;
137 }
138
    //// 检查字符设备类型。
    // 该函数仅用于下面文件打开系统调用 sys_open()，用于检查若打开的文件是 tty 终端字符设
    // 备时，需要对当前进程的设置和对 tty 表的设置。
```

```
    // 返回 0 检测处理成功，返回-1 表示失败，对应字符设备不能打开。
139 static int check_char_dev(struct m_inode * inode, int dev, int flag)
140 {
141         struct tty_struct *tty;
142         int min;                                        // 子设备号。
143
    // 只处理主设备号是 4（/dev/ttyxx 文件）或 5（/dev/tty 文件）的情况。/dev/tty 的子设备
    // 号是 0。如果一个进程有控制终端，则它是进程控制终端设备的同义名。即/dev/tty 设备是
    // 一个虚拟设备，它对应到进程实际使用的/dev/ttyxx 设备之一。对于一个进程来说，若其有
    // 控制终端，那么它的任务结构中的 tty 字段将是 4 号设备的某一个子设备号。
    // 如果打开操作的文件是 /dev/tty（即 MAJOR(dev) = 5），那么我们令 min = 进程任务结构
    // 中的 tty 字段，即取 4 号设备的子设备号。否则如果打开的是某个 4 号设备，则直接取其子
    // 设备号。如果得到的 4 号设备子设备号小于 0，那么说明进程没有控制终端，或者设备号错
    // 误，则返回 -1，表示由于进程没有控制终端，或者不能打开这个设备。
144         if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
145                 if (MAJOR(dev) == 5)
146                         min = current->tty;
147                 else
148                         min = MINOR(dev);
149                 if (min < 0)
150                         return -1;
    // 主伪终端设备文件只能被进程独占使用。如果子设备号表明是一个主伪终端，并且该打开文件
    // i 节点引用计数大于 1，则说明该设备已被其他进程使用。因此不能再打开该字符设备文件，
    // 于是返回 -1。否则，我们让 tty 结构指针 tty 指向 tty 表中对应结构项。若打开文件操作标
    // 志 flag 中不含无需控制终端标志 O_NOCTTY，并且进程是进程组首领，并且当前进程没有控制
    // 终端，并且 tty 结构中 session 字端为 0（表示该终端还不是任何进程组的控制终端），那么
    // 就允许为进程设置这个终端设备 min 为其控制终端。于是设置进程任务结构终端设备号字段
    // tty 值等于 min，并且设置对应 tty 结构的会话号 session 和进程组号 pgrp 分别等于进程的会
    // 话号和进程组号。
151                 if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1))
152                         return -1;
153                 tty = TTY_TABLE(min);
154                 if (!(flag & O_NOCTTY) &&
155                     current->leader &&
156                     current->tty<0 &&
157                     tty->session==0) {
158                         current->tty = min;
159                         tty->session= current->session;
160                         tty->pgrp = current->pgrp;
161                 }
    // 如果打开文件操作标志 flag 中含有 O_NONBLOCK（非阻塞）标志，则我们需要对该字符终端
    // 设备进行相关设置，设置为满足读操作需要读取的最少字符数为 0，设置超时定时值为 0，
    // 并把终端设备设置成非规范模式。非阻塞方式只能工作于非规范模式。在此模式下当 VMIN
    // 和 VTIME 均设置为 0 时，辅助队列中有多少支付进程就读取多少字符，并立刻返回。参见
    // include/termios.h 文件后的说明。
162                 if (flag & O_NONBLOCK) {
163                         TTY_TABLE(min)->termios.c_cc[VMIN] =0;
164                         TTY_TABLE(min)->termios.c_cc[VTIME] =0;
165                         TTY_TABLE(min)->termios.c_lflag &= ~ICANON;
166                 }
167         }
168         return 0;
169 }
```

```
170
    //// 打开（或创建）文件系统调用。
    // 参数 filename 是文件名，flag 是打开文件标志，它可取值：O_RDONLY（只读）、O_WRONLY
    // （只写）或 O_RDWR（读写），以及 O_CREAT（创建）、O_EXCL（被创建文件必须不存在）、
    // O_APPEND（在文件尾添加数据）等其他一些标志的组合，如果本调用创建了一个新文件，则
    // mode 就用于指定文件的许可属性。这些属性有 S_IRWXU（文件宿主具有读、写和执行权限）、
    // S_IRUSR（用户具有读文件权限）、S_IRWXG（组成员具有读、写和执行权限）等等。对于新
    // 创建的文件，这些属性只应用于将来对文件的访问，创建了只读文件的打开调用也将返回一
    // 个可读写的文件句柄。如果调用操作成功，则返回文件句柄(文件描述符)，否则返回出错码。
    // 参见 sys/stat.h、fcntl.h。
171 int sys_open(const char * filename,int flag,int mode)
172 {
173         struct m_inode * inode;
174         struct file * f;
175         int i,fd;
176
    // 首先对参数进行处理。将用户设置的文件模式和进程模式屏蔽码相与，产生许可的文件模式。
    // 为了为打开文件建立一个文件句柄，需要搜索进程结构中文件结构指针数组，以查找一个空
    // 闲项。空闲项的索引号 fd 即是句柄值。若已经没有空闲项，则返回出错码（参数无效）。
177         mode &= 0777 & ~current->umask;
178         for(fd=0 ; fd<NR_OPEN ; fd++)
179                 if (!current->filp[fd])                         // 找到空闲项。
180                         break;
181         if (fd>=NR_OPEN)
182                 return -EINVAL;
    // 然后我们设置当前进程的执行时关闭文件句柄（close_on_exec）位图，复位对应的比特位。
    // close_on_exec 是一个进程所有文件句柄的位图标志。每个比特位代表一个打开着的文件描
    // 述符，用于确定在调用系统调用 execve()时需要关闭的文件句柄。当程序使用 fork()函数
    // 创建了一个子进程时，通常会在该子进程中调用 execve()函数加载执行另一个新程序。此时
    // 子进程中开始执行新程序。若一个文件句柄在 close_on_exec 中的对应比特位被置位，那么
    // 在执行 execve()时该对应文件句柄将被关闭，否则该文件句柄将始终处于打开状态。当打开
    // 一个文件时，默认情况下文件句柄在子进程中也处于打开状态。因此这里要复位对应比特位。
    // 然后为打开文件在文件表中寻找一个空闲结构项。我们令 f 指向文件表数组开始处。搜索空
    // 闲文件结构项（引用计数为 0 的项），若已经没有空闲文件表结构项，则返回出错码。另外，
    // 第 184 行上的指针赋值 "0+file_table " 等同于 "file_table" 和 "&file_table[0]"。
    // 不过这样写可能更能明了一些。
183         current->close_on_exec &= ~(1<<fd);
184         f=0+file_table;
185         for (i=0 ; i<NR_FILE ; i++,f++)
186                 if (!f->f_count) break;
187         if (i>=NR_FILE)
188                 return -EINVAL;
    // 此时我们让进程对应文件句柄 fd 的文件结构指针指向搜索到的文件结构，并令文件引用计数
    // 递增 1。然后调用函数 open_namei()执行打开操作，若返回值小于 0，则说明出错，于是释放
    // 刚申请到的文件结构，返回出错码 i。若文件打开操作成功，则 inode 是已打开文件的 i 节点
    // 指针。
189         (current->filp[fd]=f)->f_count++;
190         if ((i=open_namei(filename,flag,mode,&inode))<0) {
191                 current->filp[fd]=NULL;
192                 f->f_count=0;
193                 return i;
194         }
    // 根据已打开文件 i 节点的属性字段，我们可以知道文件的类型。对于不同类型的文件，我们
```

```
    // 需要作一些特别处理。 如果打开的是字符设备文件，那么我们就要调用 check_char_dev()
    // 函数来检查当前进程是否能打开这个字符设备文件。如果允许（函数返回 0），那么在
    // check_char_dev()中会根据具体文件打开标志为进程设置控制终端。 如果不允许打开
    // 使用该字符设备文件，那么我们只能释放上面申请的文件项和句柄资源，返回出错码。
195 /* ttys are somewhat special (ttyxx major==4, tty major==5) */
    /* ttys 有些特殊（ttyxx 的主设备号==4，tty 的主设备号==5）*/
196         if (S_ISCHR(inode->i_mode))
197                 if (check_char_dev(inode,inode->i_zone[0],flag)) {
198                         iput(inode);
199                         current->filp[fd]=NULL;
200                         f->f_count=0;
201                         return -EAGAIN;          // 出错号：资源暂时不可用。
202                 }
    // 如果打开的是块设备文件，则检查盘片是否更换过。若更换过则需要让高速缓冲区中该设备
    // 的所有缓冲块失效。
203 /* Likewise with block-devices: check for floppy_change */
    /* 同样对于块设备文件：需要检查盘片是否被更换 */
204         if (S_ISBLK(inode->i_mode))
205                 check_disk_change(inode->i_zone[0]);
    // 现在我们初始化打开文件的文件结构。设置文件结构属性和标志，置句柄引用计数为 1，并
    // 设置 i 节点字段为打开文件的 i 节点，初始化文件读写指针为 0。最后返回文件句柄号。
206         f->f_mode = inode->i_mode;
207         f->f_flags = flag;
208         f->f_count = 1;
209         f->f_inode = inode;
210         f->f_pos = 0;
211         return (fd);
212 }
213
    //// 创建文件系统调用。
    // 参数 pathname 是路径名，mode 与上面的 sys_open()函数相同。
    // 成功则返回文件句柄，否则返回出错码。
214 int sys_creat(const char * pathname, int mode)
215 {
216         return sys_open(pathname, O_CREAT | O_TRUNC, mode);
217 }
218
    // 关闭文件系统调用。
    // 参数 fd 是文件句柄。
    // 成功则返回 0，否则返回出错码。
219 int sys_close(unsigned int fd)
220 {
221         struct file * filp;
222
    // 首先检查参数有效性。若给出的文件句柄值大于程序同时能打开的文件数 NR_OPEN，则返回
    // 出错码（参数无效）。然后复位进程的执行时关闭文件句柄位图对应位。若该文件句柄对应
    // 的文件结构指针是 NULL，则返回出错码。
223         if (fd >= NR_OPEN)
224                 return -EINVAL;
225         current->close_on_exec &= ~(1<<fd);
226         if (!(filp = current->filp[fd]))
227                 return -EINVAL;
    // 现在置该文件句柄的文件结构指针为 NULL。若在关闭文件之前，对应文件结构中的句柄引用
```

```
    // 计数已经为 0，则说明内核出错，停机。否则将对应文件结构的引用计数减 1。此时如果它还
    // 不为 0，则说明有其他进程正在使用该文件，于是返回 0（成功）。如果引用计数已等于 0，
    // 说明该文件已经没有进程引用，该文件结构已变为空闲。则释放该文件 i 节点，返回 0。
228         current->filp[fd] = NULL;
229         if (filp->f_count == 0)
230                 panic("Close: file count is 0");
231         if (--filp->f_count)
232                 return (0);
233         iput(filp->f_inode);
234         return (0);
235 }
236
```